Expressiveness of Generic Process Shape Types

Jan Jakubův J. B. Wells

Heriot-Watt University

March 31, 2010 [*]

Abstract

Shape types are a general concept of process types which work for many process calculi. We extend the previously published Poly✶ system of shape types to support name restriction. We evaluate the expressiveness of the extended system by showing that shape types are more expressive than an implicitly typed π-calculus and an explicitly typed Mobile Ambients. We demonstrate that the extended system makes it easier to enjoy advantages of shape types which include polymorphism, principal typings, and a type inference implementation.

1 Introduction

Many type systems for many process calculi have been developed to statically guarantee various important properties of processes. Types differ among these systems and their properties, such as soundness, have to be proved separately for each system. *Shape types* are a general concept of polymorphic process types which can express and verify various properties of processes. Poly✶ [12, 11] is a general framework which, for a wide range of process calculi, can be instantiated to make ready-to-use sound type systems which use shape types. Only rewriting rules satisfying common syntactic conditions are needed for instantiating Poly✶.

Many process calculi share semantically equivalent constructions, such as, *parallel composition* ("$|$"), prefixing a process with an action (sometimes called a *capability*) ("."), and *name restriction* ("ν"). Specific calculi differ mainly in the syntax and semantics of actions (capabilities). Meta✶ [12, 11] is metacalculus which fixes semantics of the shared constructions and provides a way to describe syntax and semantics of actions by a description $\mathcal{R}$ of rewriting rules. Given $\mathcal{R}$, Meta✶ makes the calculus $C_{\mathcal{R}}$ and Poly✶ makes the type system $S_{\mathcal{R}}$ for $C_{\mathcal{R}}$. $\mathcal{R}$ can describe many calculi including, e.g., the π-calculus, Mobile Ambients, numerous variations of these, and other systems. All instantiations of Poly✶ share *shape predicates* which describe allowed syntactic configurations of Meta✶ processes. *Shape ($\mathcal{R}$-)types* of $S_{\mathcal{R}}$ are shape predicates whose meaning is guaranteed by a simple test to be closed under rewriting with $\mathcal{R}$. Every $S_{\mathcal{R}}$ has desirable properties such as subject reduction, the existence of principal typings [17], and an already implemented type inference algorithm[1].

1.1 Contributions

This paper extends the POLY✶ system to support name restriction and also proves POLY✶ shape types are more expressive than some previous systems for specific calculi. The contributions are as follows. (1) Sec. 2 presents the extended POLY✶ system. Sections 3, 4 show (2) how to easily use shape types with well-known calculi (the π-calculus [14, 13], Mobile Ambients [3]), (3) demonstrate polymorphic abilities of shape types, and (4) prove that shape types are more expressive than predicates of two type systems (*implicitly* typed π-calculus [16], *explicitly* typed Mobile Ambients [4]) custom designed for the above calculi. Finally, (5) we advocate a generic notion of shape types and show that they can be used instead of predicates of many other systems. We consider contributions (4) & (5) to be the main contribution of the paper.

Contribution (2) shows how to use POLY✶ and shape types without needing to fully understand all the details of the underlying formalism. Thus it helps to bridge over the problem of complexity of POLY✶ which is inevitably implied by its high generality and which has been daunting to some readers of earlier papers. Contribution (3) shows an aspect of shape types which is not common for other systems. An accompanying technical report [9] (TR), which extends this paper and contains proofs of main theorems, additionally shows how to use shape types for *flow analysis* of BioAmbients and proves its superior expressiveness to an earlier flow analysis system [15]. This work was left out for space reasons. For all the three systems we have proven not only that shape types are more expressive but also that they can be used to achieve exactly the same results as the original systems which might be important for some of their applications. We believe that the diversity of the mentioned systems and their intended applications provides a reasonable justification for contribution (5).

1.2 Notations and Preliminaries

Let i, j, k range over natural numbers. $\mathcal{P}_{\mathrm{fin}}(U)$ is the set of all finite subsets of a set U, "\" denotes set subtraction. Let $u \mapsto v$ be an alternate pair notation used in functions. $f[u \mapsto v]$ stands for the function that maps u to v and other values as f. Moreover, $U \to V$ ($U \to_{\mathrm{fin}} V$) is the set of all (all finite) functions f with $\mathsf{dom}(f) \subseteq U$ and $\mathsf{rng}(f) \subseteq V$.

[1] `http://www.macs.hw.ac.uk/ultra/polystar` (includes a web demonstration)

$$
\begin{array}{rll}
a, b \in & \mathsf{BasicName} & ::= \mathsf{a} \mid \mathsf{b} \mid \cdots \mid \mathsf{in} \mid \mathsf{out} \mid \mathsf{open} \mid \cdots \mid \texttt{[]} \mid \bullet \mid \cdots \\
x, y \in & \mathsf{Name} & ::= a^i \\
F \in & \mathsf{Form} & ::= x_0 \ldots x_k \\
M \in & \mathsf{Message} & ::= F \mid \mathsf{0} \mid M_0.M_1 \\
E \in & \mathsf{Element} & ::= x \mid \texttt{(}x_1, \ldots, x_k\texttt{)} \mid \texttt{<}M_1, \ldots, M_k\texttt{>} \\
A \in & \mathsf{Action} & ::= E_0 \ldots E_k \\
P, Q \in & \mathsf{Process} & ::= \mathsf{0} \mid A.P \mid (P \,|\, Q) \mid \nu x.P \mid !P
\end{array}
$$

Figure 1: Syntax of META✶ processes.

$$
\begin{array}{ccc}
P \,|\, Q \equiv Q \,|\, P & P \,|\, (Q \,|\, R) \equiv (P \,|\, Q) \,|\, R & P \,|\, \mathsf{0} \equiv P \\
\mathsf{0} \equiv !\mathsf{0} & \nu x.\nu y.P \equiv \nu y.\nu x.P & !P \equiv P \,|\, !P
\end{array}
$$

$$
A.\nu x.P \equiv \nu x.A.P \text{ if } x \notin \mathsf{fn}(A) \cup \mathsf{bn}(A) \qquad P \,|\, \nu x.Q \equiv \nu x.(P \,|\, Q) \text{ if } x \notin \mathsf{fn}(P)
$$

Figure 2: META✶ structural equivalence (structural rules omitted).

2 Metacalculus META✶ and Generic Type System POLY✶

2.1 General Syntax of Processes

META✶ process syntax, presented in Fig. 1, allows embeddings of many calculi. A name a^i is a pair of a *basic name* a and a natural number i. The basic part of a name x is denoted $\underline{x}$, that is, $\underline{a^i} = a$. When α-converting, we preserve the basic name and change the number. We write a instead of a^0 when no confusion can arise.

Processes are built from the null process "$\mathsf{0}$" by prefixing with an action ("."), by parallel composition ("|"), by name restriction ("ν"), and by replication ("!"). Actions can encode prefixes from various calculi such as π-calculus communication actions, Mobile Ambients capabilities, or ambient boundaries. The abbreviation "$x_1 \ldots x_k\,\texttt{[}P\texttt{]}$", which further supports ambient syntax, stands for "$x_1 \ldots x_k\,\texttt{[]}.P$" ($\texttt{[]}$ is a single name).

Process constructors have standard semantics. "$\mathsf{0}$" is an inactive process, "$A.P$" executes the action A and continues as P, "$P \,|\, Q$" runs P and Q in parallel, "$\nu x.P$" behaves as P with private name x (i.e., x differs from all names outside P), and "$!P$" acts as infinitely many copies of P in parallel ("$P \,|\, P \,|\, \cdots$"). Let "." and "ν" bind more tightly than "|". These constructors have standard properties given by structural equivalence $\equiv$ (Fig. 2), e.g., "|" is commutative, adjacent "ν" can be interchanged, etc. In contrast, the semantics of actions is defined by instantiating META✶ (see below). Currently, META✶ does not support the choice operator "+" as a built in primitive. However, "$P + Q$" can be encoded as "$\mathsf{ch}.(P \,|\, Q)$" provided rewriting rules are extended to use this encoding.

All occurrences x in "$\nu x.P$" are (ν-)bound. When the action A contains an element "$\texttt{(}x_1, \ldots, x_k\texttt{)}$" then all occurrences of the x_i's in "$A.P$" as well as in A on its own are called (input-)bound. An occurrence of x that is not bound is free. The occurrence of a in a^i is bound (resp. free) when this occurrence of a^i is. A bound occurrence of a^i can be α-converted only to a^j with a the same. We identify α-convertible processes. The set of free names of P is denoted $\mathsf{fn}(P)$. The set $\mathsf{fbn}(P)$ (resp. $\mathsf{ibn}(P)$, $\mathsf{nbn}(P)$) contains free (resp. input-bound, ν-bound) basic names of P. The set of bound names of A is written $\mathsf{bn}(A)$.

A process P is *well scoped* when (W1) $\mathsf{fbn}(P)$, $\mathsf{ibn}(P)$, and $\mathsf{nbn}(P)$ do not overlap, (W2) nested input binders do not bind the same basic name, and (W3) no action contains an input-binding of a basic name more than once. These conditions are important for type inference. We allow only well scoped processes.

A Meta✶ substitution σ is a finite function from Name to $\mathsf{Message}$. Application of σ to P, written $P\sigma$, behaves as usual except the following. (1) It places a special name "$\bullet$" at positions that would otherwise be syntax errors (e.g., $(\mathsf{in}\ \mathsf{x}.0)\{\mathsf{x} \mapsto \mathsf{out}\ \mathsf{b}\} = \mathsf{in}\ \bullet.0$). (2) When a composed message M is substituted for a single name action x in "$\mathsf{x}.P$", then M's components are pushed from right to left onto $P\sigma$ (e.g., $(\mathsf{x}.0)\{\mathsf{x} \mapsto (\mathsf{a}.\mathsf{b}).\mathsf{c}\} = \mathsf{a}.\mathsf{b}.\mathsf{c}.0$). The full definition of $P\sigma$ is in the TR.

2.2 Instantiations of Meta✶

Meta✶ provides syntax to describe rewriting rules that give meaning to actions and also defines how these rules yield a rewriting relation on processes. The syntax is best explained by an example. The following rule description (in which "$\mathtt{\{}\mathring{\mathtt{x}}\ \mathtt{:=}\ \mathring{\mathtt{n}}\mathtt{\}}\mathring{\mathtt{Q}}$" describes substitution application)

$$\textbf{rewrite}\{\ \mathring{\mathtt{c}}\mathtt{<}\mathring{\mathtt{n}}\mathtt{>.}\mathring{\mathtt{P}} \mid \mathring{\mathtt{c}}\mathtt{(}\mathring{\mathtt{x}}\mathtt{).}\mathring{\mathtt{Q}} \hookrightarrow \mathring{\mathtt{P}} \mid \mathtt{\{}\mathring{\mathtt{x}}\ \mathtt{:=}\ \mathring{\mathtt{n}}\mathtt{\}}\mathring{\mathtt{Q}}\ \}$$

directly corresponds to the standard π-calculus communication rule "$c\mathtt{<}n\mathtt{>}.P \mid c(x).Q \Rightarrow P \mid Q\{x \mapsto n\}$". The circle-topped letters stand at the place of name, message, and process metavariables. Given a set $\mathcal{R}$ of rule descriptions in the above syntax, Meta✶ automatically infers the rewriting relation $\overset{\mathcal{R}}{\hookrightarrow}$ which incorporates structural equivalence and congruence rules (e.g., "$P \overset{\mathcal{R}}{\hookrightarrow} Q \Rightarrow \nu x.P \overset{\mathcal{R}}{\hookrightarrow} \nu x.Q$"). A rules description instantiates Meta✶ to a particular calculus, e.g., the set $\mathcal{R}$ containing only the above rule description instantiates Meta✶ to the π-calculus.

Further examples of Meta✶ instantiations are given in Sec. 3.3 and 4.3. A rule description can also contain a concrete Meta✶ name (e.g. "out") when an exact match is required. We require that these names are never bound in any process. Complete definitions of the syntax of rewriting rules and of the rewriting relation $\overset{\mathcal{R}}{\hookrightarrow}$ is left to the TR [9, Sec. 2.2].

2.3 Poly✶ Shape Predicates and Types for Meta✶

A *shape predicate* describes possible structures of process syntax trees. When a rewriting rule from $\mathcal{R}$ is applied to a process, its syntax tree changes, and sometimes the new syntax tree no longer satisfies the same shape predicates. All Poly✶ ($\mathcal{R}$-)types are shape predicates that describe process sets closed

Syntax of Poly✶ *shape predicates:*

$$\begin{array}{rlll@{\qquad}rlll}
\varphi \in & \textsf{FormType} & ::= & a_0 \ldots a_k & \alpha \in & \textsf{ActionType} & ::= & \varepsilon_0\, \varepsilon_1 \ldots \varepsilon_k \\
\Phi \in & \textsf{FormTypeSet} & = & \mathcal{P}_{\text{fin}}(\textsf{FormType}) & \chi \in & \textsf{Node} & ::= & \texttt{X} \mid \texttt{Y} \mid \texttt{Z} \mid \cdots \\
\mu \in & \textsf{MessageType} & ::= & \Phi\ast \mid a & \eta \in & \textsf{Edge} & ::= & \chi_0 \xrightarrow{\alpha} \chi_1 \\
\varepsilon \in & \textsf{ElementType} & ::= & a \mid \texttt{(}a_1, \ldots, a_k\texttt{)} \mid & G \in & \textsf{ShapeGraph} & = & \mathcal{P}_{\text{fin}}(\textsf{Edge}) \\
& & & \texttt{<}\mu_1, \ldots, \mu_k\texttt{>} & \pi \in & \textsf{ShapePredicate} & ::= & \langle G, \chi \rangle
\end{array}$$

Rules for matching Meta✶ *entities against shape predicates:*

$$\begin{array}{lll}
\vdash a^i : a & \vdash \texttt{(}a_1^{i_1}, \ldots, a_k^{i_k}\texttt{)} : \texttt{(}a_1, \ldots, a_k\texttt{)} & (\vdash M_0 : \Phi \;\&\vdash M_1 : \Phi) \Rightarrow \vdash M_0.M_1 : \Phi \\
\vdash \textsf{0} : \Phi & (\vdash F : \varphi \;\&\; \varphi \in \Phi) \Rightarrow \vdash F : \Phi & (M \notin \textsf{Name} \;\&\vdash M : \Phi) \Rightarrow \vdash M : \Phi\ast
\end{array}$$

$$\begin{array}{l}
(\forall i \leq k \colon \vdash E_i : \varepsilon_i) \Rightarrow \vdash E_0 \ldots E_k : \varepsilon_0 \ldots \varepsilon_k \\
(\forall i \colon 0 < i \leq k \;\&\vdash M_i : \mu_i) \Rightarrow \vdash \texttt{<}M_1, \ldots, M_k\texttt{>} : \texttt{<}\mu_1, \ldots, \mu_k\texttt{>}
\end{array}$$

$$\begin{array}{ll}
\vdash \textsf{0} : \pi & \\
\vdash P : \pi \Rightarrow \vdash \nu x.P : \pi & (\vdash P : \pi \;\&\vdash Q : \pi) \Rightarrow \vdash P \mid Q : \pi \\
\vdash P : \pi \Rightarrow \vdash !P : \pi & ((\chi_0 \xrightarrow{\alpha} \chi_1) \in G \;\&\vdash A : \alpha \;\&\vdash P : \langle G, \chi_1 \rangle) \Rightarrow \vdash A.P : \langle G, \chi_0 \rangle
\end{array}$$

Figure 3: Syntax and semantics of Poly✶ shape predicates.

under rewriting using $\mathcal{R}$. For feasibility, types are defined via a syntactic test that enforces rewriting-closedness. Intuitively, the syntactic test tries to apply the rules from $\mathcal{R}$ to all active positions in a shape graph and checks whether all the edges newly generated by this application are already present in the graph. Further restrictions are used to ensure the existence of principal typings.

Fig. 3 defines shape predicate syntax. Action types are similar to actions except that action types are built from basic names instead of names, and compound messages are described up to commutativity, associativity, and repetitions of their parts. Thus an action type describes a set of actions. A shape predicate $\langle G, \chi \rangle$ is a directed finite graph with root χ and with edges labeled by action types. A process P matches π when P's syntax tree is a "subgraph" of π. Shape predicate can have loops and thus describe syntax trees of arbitrary height.

Fig. 3 also describes matching Meta✶ entities against shape predicates. The rule matching actions against action types also matches forms against form types. Matching entities against types does not depend on $\mathcal{R}$, i.e., it works the same in any Meta✶ instantiation. The *meaning* $[\![\pi]\!]$ of the shape predicate π is the set $\{P | \vdash P : \pi\}$ of all processes matching π.

A shape predicate π is *semantically closed* w.r.t. a rule set $\mathcal{R}$ when $[\![\pi]\!]$ is closed under $\mathcal{R}$-rewritings, i.e., when $\vdash P : \pi$ and $P \overset{\mathcal{R}}{\hookrightarrow} Q$ imply $\vdash Q : \pi$ for any P and Q. Because deciding semantic closure w.r.t. an arbitrary $\mathcal{R}$ is nontrivial, we use an easier-to-decide property, namely *syntactic closure*, which by design is algorithmically verifiable. *$\mathcal{R}$-types* are shape predicates syntactically closed w.r.t. $\mathcal{R}$. A type π of P is a *principal typing* of P when $[\![\pi]\!] \subseteq [\![\pi_0]\!]$ for any other type π_0 of P. There are *width* and *depth* restrictions to ensure principal typings. Details are left to our TR [9, Sec. 2.4].

2.4 Proving Greater Expressiveness of Poly✶

We now discuss how to consider some process calculus C and its type system S_C and prove the greater expressiveness of the related Meta✶ and Poly✶ instantiations. Sections 3 and 4 follow this approach. Usually S_C defines predicates (ranged over by φ) which represent properties of processes (ranged over by B) of C. Then S_C defines the relation $\rhd B : \varphi$ which represents statements "B has the property φ" and which is preserved under rewriting of B in C. The Meta✶ description $\mathcal{R}$ of C's rewriting rules gives us the calculus $C_{\mathcal{R}}$ and its shape type system $S_{\mathcal{R}}$.

Firstly we need to set up a correspondence between C and $C_{\mathcal{R}}$, that is, we need an encoding $\llparenthesis \cdot \rrparenthesis$ of processes B into Meta✶ which preserves C's rewriting relation $\rightarrow$. The following property, which is usually easy to prove, formulates this modulo $\equiv$ because structural equivalences of different calculi might differ.

Property 2.1 *When $B_0 \rightarrow B_1$ then $\exists B'_0, B'_1$ such that $B_0 \equiv B'_0$ & $\llparenthesis B'_0 \rrparenthesis \overset{\mathcal{R}}{\hookrightarrow} \llparenthesis B'_1 \rrparenthesis$ & $B'_1 \equiv B_1$. When $\llparenthesis B_0 \rrparenthesis \overset{\mathcal{R}}{\hookrightarrow} P_1$ then $\exists B_1$ such that $B_0 \rightarrow B_1$ & $\llparenthesis B_1 \rrparenthesis \equiv P_1$.*

Predicates φ of S_C are commonly preserved under renaming of bound basic names, that is, $\rhd(\nu x)B : \varphi$ usually implies $\rhd(\nu a^0)(B\{x \mapsto a^0\}) : \varphi$ (for a not in B). Predicates of similar systems can not be directly translated to Poly✶ shape types with the corresponding meaning because shape types do not have this property. In other words, the difference in handling of bound names between Poly✶ and other systems makes some straightforward embeddings impossible.

We investigate two reasonable ways to embed S_C in $S_{\mathcal{R}}$, that is, to decide $\rhd B : \varphi$ using $S_{\mathcal{R}}$'s relation "$\vdash$". (1) In Sec. 4.4 about Mobile Ambients, we translate φ together with information about bound basic names of B into a shape type. (2) In Sec. 3.4 about the π-calculus, we show how to decide $\rhd B : \varphi$ by a simple check on a principal shape type of B. The fact that both embeddings of predicates φ depend on a process B is not a limitation because B is known for desirable applications like type checking.

We stress that these embeddings serve the theoretical purpose of proving greater expressiveness and are not necessary for a practical use of shape types. When S_C is designed to verify a certain fixed property of processes which can be expressed as a property of shape types, then we can use $S_{\mathcal{R}}$ directly for the same purposes as S_C without any embedding. We show how to do this for the two systems in Sec. 3.3 and 4.3. We can also design a property of processes directly on shape types without any reference to another analysis system. Our TR [9, Sec. 3] discusses this further.

2.5 Discussion

Poly✶ presented above extends the previously published Poly✶ [12] with name restriction. The previously published system [12] supports restriction only in Meta✶ but no processes with ν are typable in Poly✶ instantiations. An earlier attempt in a technical report [11] to handle name restriction was found incon-

Syntax of the π-calculus processes:

$$\begin{aligned} c,n,m \in{}& \mathsf{PiName} &=&\ \mathsf{Name}\setminus\{\bullet\} \\ N \in{}& \mathsf{PiAction} &::=&\ c\texttt{(}n_1,\ldots,n_k\texttt{)} \mid c\texttt{<}n_1,\ldots,n_k\texttt{>} \\ B \in{}& \mathsf{PiProcess} &::=&\ \mathsf{0} \mid (B_0 \mathrel{\mathsf{I}} B_1) \mid N.B \mid !B \mid (\nu n)B \end{aligned}$$

Rewriting relation of the π-calculus (≡ is standard defined in TR [9, Fig. 8]):

$$c\texttt{(}n_1,\ldots,n_k\texttt{)}.B_0 \mathrel{\mathsf{I}} c\texttt{<}m_1,\ldots,m_k\texttt{>}.B_1 \to B_0\{n_1 \mapsto m_1,\ldots,n_k \mapsto m_k\} \mathrel{\mathsf{I}} B_1$$

$$B_0 \to B_1 \;\Rightarrow\; (\nu n)B_0 \to (\nu n)B_1 \qquad B_0' \equiv B_0 \;\&\; B_0 \to B_1 \;\&\; B_1 \equiv B_1' \;\Rightarrow\; B_0' \to B_1'$$

$$B_0 \to B_1 \;\Rightarrow\; B_0 \mathrel{\mathsf{I}} B_2 \to B_1 \mathrel{\mathsf{I}} B_2$$

Figure 4: The syntax and semantics of the π-calculus.

sistent [8, Sec. 3.2-4] and furthermore inadequate [8, Sec. 4] to carry out the proofs of greater expressiveness in sections 3 and 4.

The difficulty with name restriction is because a shape type represents a syntactic structure of a process, and thus presence of bound names in a process has to be somehow reflected by a shape graph. Because bound names can be α-renamed, Poly✶ needs to establish a connection between positions in a process and a shape graph which is preserved by α-conversion. This connection is provided by basic names which are the key concept of name restriction handling in this paper. For example, for the action "a<a>" there is the corresponding action type "a<a>" in its shape type. When the name a were ν-bound and α-renamed to some other name then the correspondence between the action in the process and the action type would be lost. This problem is solved by building shape types from basic names which are preserved under α-conversion.

The handling of input-bound names in the previous Poly✶ was reached by disabling their α-conversion which is possible under certain circumstances. But α-conversion of ν-bound names can not be avoided and thus a new approach has been developed.

3 Shape Types for the π-calculus

3.1 A Polyadic π-calculus

The π-calculus [14, 13] is a process calculus involving process mobility developed by Milner, Parrow, and Walker. Mobility is abstracted as channel-based communication whose objects are atomic names. Channel labels are not distinguished from names and can be passed by communication. This ability, referred as *link passing*, is the π-calculus feature that most distinguishes it from its predecessors. We use a polyadic version of the π-calculus which supports communication of tuples of names.

Fig. 5 presents the syntax and semantics of the π-calculus. Processes are built from Meta✶ names. The process "$c\texttt{(}n_1,\ldots,n_k\texttt{)}.B$", which (input)-binds the names n_i's, waits to receive a k-tuple of names over channel c and then behaves like B with the received values substituted for n_i's. The process "$c\texttt{<}n_1,\ldots,n_k\texttt{>}.B$" sends the k-tuple n_1, …, n_k over channel c and then be-

Syntax of Tpi *types:*

$$\begin{array}{rlcl}
\beta \in & \mathsf{PiTypeVariable} & ::= & \imath \mid \imath' \mid \imath'' \mid \cdots \\
\delta \in & \mathsf{PiType} & ::= & \beta \mid \uparrow[\delta_1, \ldots, \delta_k] \\
\Delta \in & \mathsf{PiContext} & = & \mathsf{BasicName} \to_{\mathrm{fin}} \mathsf{PiType}
\end{array}$$

Typing rules of Tpi*:*

$$\begin{array}{ll}
\Delta \vdash \mathsf{0} & \Delta \vdash B_0 \;\&\; \Delta \vdash B_1 \;\Rightarrow\; \Delta \vdash B_0 \mid B_1 \\
\Delta \vdash B \;\Rightarrow\; \Delta \vdash !B & \Delta[\underline{n} \mapsto \delta] \vdash B \;\Rightarrow\; \Delta \vdash (\nu n)B
\end{array}$$

$$\Delta(\underline{c}) = \uparrow[\delta_1, \ldots, \delta_k] \;\&\; \Delta[\underline{n_1} \mapsto \delta_1, \ldots, \underline{n_k} \mapsto \delta_k] \vdash B \;\Rightarrow\; \Delta \vdash c(n_1, \ldots, n_k).B$$

$$\Delta(\underline{c}) = \uparrow[\Delta(\underline{n_1}), \ldots, \Delta(\underline{n_k})] \;\&\; \Delta \vdash B \;\Rightarrow\; \Delta \vdash c\texttt{<}n_1, \ldots, n_k\texttt{>}.B$$

Figure 5: Syntax of Tpi types and typing rules.

haves like B. Other constructors have the meaning as in Meta✶ (Sec. 2.1). The sets of names $\mathsf{fn}(B)$, $\mathsf{fbn}(B)$, $\mathsf{ibn}(B)$, $\mathsf{nbn}(B)$ are defined as in Meta✶.

Processes are identified up to α-conversion of bound names which preserves basic names. A substitution in the π-calculus is a finite function from names to names, and its application to B is written postfix, e.g., "$B\{n \mapsto m\}$". A process B is *well scoped* when (S1) $\mathsf{fbn}(B)$, $\mathsf{ibn}(B)$, and $\mathsf{nbn}(B)$ do not overlap, (S2) nested input binders do not bind the same basic name, and (S3) no input action contains the same basic name more then once. Henceforth, we require processes to be well scoped (well-scopedness is preserved by rewriting).

Example 3.1 *Let* $B = \mathsf{!s(x,y).x\texttt{<}y\texttt{>}.0 \mid s\texttt{<}a,n\texttt{>}.0 \mid a(v).v(p).0 \quad \mid n\texttt{<}o\texttt{>}.0 \mid}$
$\mathsf{\mid s\texttt{<}b,m\texttt{>}.0 \mid b(w).v(q,r).0 \mid m\texttt{<}o,o\texttt{>}.0}$
Using the rewriting relation $\to$ *sequentially four times we can obtain (among others) the process* "$\mathsf{!s(x,y).x\texttt{<}y\texttt{>}.0 \mid n(p).0 \mid n\texttt{<}o\texttt{>}.0 \mid m(q,r).0 \mid m\texttt{<}o,o\texttt{>}.0}$".

3.2 Types for the Polyadic π-calculus (Tpi)

We compare Poly✶ with a simple type system [16, Ch. 3] for the polyadic π-calculus presented by Turner which we name Tpi. Tpi is essentially Milner's sort discipline [13]. In the polyadic settings, an arity mismatch error on channel c can occur when the lengths of the sent and received tuple do not agree, like in "$c(n).\mathsf{0} \mid c\texttt{<}m,m\texttt{>}.\mathsf{0}$". Processes which can never evolve to a state with a similar situation are called *communication safe*. Tpi verifies communication safety of π-processes.

The syntax and typing rules of Tpi are presented in Fig. 5. Recall that $\underline{n}$ denotes the basic name of n. Types δ are assigned to names. Type variables β are types of names which are not used as channel labels. The type "$\uparrow[\delta_1, \ldots, \delta_k]$" describes a channel which can be used to communicate any k-tuple whose i-th name has type δ_i. A context Δ assigns types to free names of a process (via their basic names). The relation $\Delta \vdash B$, which is preserved under rewriting, expresses that the actual usage of channels in B agrees with Δ. When $\Delta \vdash B$ for some Δ then B is communication safe. The opposite does not necessarily hold.

Example 3.2 *Given B from Ex. 3.1 we can see that there is no Δ such that $\Delta \vdash B$. It is because the parts* `s<a,n>` *and* `s<a,m>` *imply that types of* `n` *and* `m` *must be equal while the parts* `n<o>` *and* `m<o,o>` *force them to be different. On the other hand B is communication safe. We check this using* Poly✶ *in Sec 3.3.*

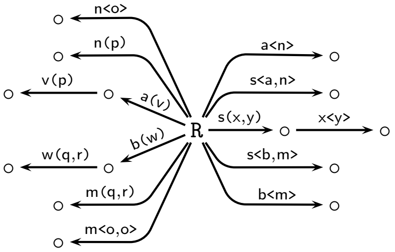
3.3 Instantiation of Meta✶ to the π-calculus

The π-calculus syntax from Sec. 3.1 already matches the Meta✶ syntax and thus only the following $\mathcal{P}$ is needed to instantiate Meta✶ to the calculus $C_{\mathcal{P}}$ and Poly✶ to its type system $S_{\mathcal{P}}$. Sec. 3.4 shows that $C_{\mathcal{P}}$ is essentially identical to the above π-calculus.

$$\mathcal{P} = \bigcup_{k=0}^{\infty} \left\{ \textbf{rewrite}\{\, \mathring{\mathtt{c}}\mathtt{<}\mathring{\mathtt{M}}_1,\ldots,\mathring{\mathtt{M}}_k\mathtt{>.}\mathring{\mathtt{P}} \mid \mathring{\mathtt{c}}\mathtt{(}\mathring{\mathtt{a}}_1,\ldots,\mathring{\mathtt{a}}_k\mathtt{).}\mathring{\mathtt{Q}} \;\hookrightarrow \mathring{\mathtt{P}} \mid \{\mathring{\mathtt{a}}_1 := \mathring{\mathtt{M}}_1,\ldots,\mathring{\mathtt{a}}_k := \mathring{\mathtt{M}}_k\}\mathring{\mathtt{Q}}\,\} \right\}$$

Each communication prefix length has its own rule; in our implementation, a single rule can uniformly handle all lengths, but the formal Meta✶ presentation is deliberately simpler. The next example shows how to check communication safety in $S_{\mathcal{P}}$ without using Tpi.

Example 3.3. Let P be a Meta✶ equivalent of B from Ex. 3.1. We can compute a principal $\mathcal{P}$-type π_P of P which is displayed on the right. Node `R` is its root. The type π_P contains all computational futures of P in one place. Thus, because there are no two edges from `R` labeled by "a`(`$b_1,\ldots,b_k$`)`" and "a`<`$b'_1,\ldots,b'_j$`>`" with $k \neq j$, we can conclude that P is communication safe which Ex. 3.2 shows Tpi can not do. Our implementation can be instructed (using an additional rule) to insert the error name • at the place of communication errors. Any type of P without • then implies P's communication safety.



3.4 Embedding of Tpi in Poly✶

Using the terminology from Sec. 2.4 we have that C is the π-calculus, S_C is Tpi, predicates φ of S_C are contexts Δ, and S_C's relation $\rhd B : \varphi$ is $\Delta \vdash B$. Moreover $\mathcal{R}$ is $\mathcal{P}$ which was introduced with $C_{\mathcal{P}}$ and $S_{\mathcal{P}}$ in Sec. 3.3. This section provides a formal comparison which shows how to, for a given B and Δ, answer the question $\Delta \vdash B$ using $S_{\mathcal{P}}$.

As stated in Sec. 2.4, to relate Tpi and $S_{\mathcal{P}}$ we need to provide an encoding $(\!|\cdot|\!)$ of π-processes in Meta✶. This $(\!|\cdot|\!)$, found in TR [9, Fig. 10] , is almost an identity because the π-calculus syntax (Fig. 4) already agrees with Meta✶. Thus $(\!|\cdot|\!)$ mainly changes the syntactic category. Prop. 2.1 holds in the above context.

Given Δ, we define a shape type property which holds for the principal type π_B of $(\!|B|\!)$ iff $\Delta \vdash B$. The property is given by the relation $\Delta \cong \pi$ from Fig. 6.

The set of expected and actual channel types of G:

$$\mathsf{chtypes}(\Delta,G)=\{(\Delta(a),\uparrow[\Delta(b_1),\ldots,\Delta(b_k)]):(\chi\xrightarrow{a\,(b_1,\ldots,b_k)}\chi')\in G\vee(\chi\xrightarrow{a<b_1,\ldots,b_k>}\chi')\in G\}$$

Context Δ and shape type π agreement relation $\cong$:

Write $\Delta \cong \langle G, \chi\rangle$ when there is some Δ' with the domain disjoint from Δ such that $\mathsf{chtypes}(\Delta \cup \Delta', G)$ is defined and is an identity.

Figure 6: Property of shape types corresponding to $\vdash$ of TPI.

The set $\mathsf{chtypes}(\Delta, G)$ contains pairs of TPI types extracted from G. Each pair corresponds to an edge of G labeled by an action type "$a\,(b_1, \ldots, b_k)$" or "$a<b_1, \ldots, b_k>$". The first member of the pair is a's type expected by Δ, and the second member computes a's actual usage from the types of b_i's. The set $\mathsf{chtypes}(\Delta, G)$ is undefined when some required value of Δ is not defined. The context Δ' from the definition of $\cong$ provides types of names originally bound in B. These are not mentioned by Δ but are in G. The following theorem shows how to answer $\Delta \vdash B$ by $\cong$.

Theorem 3.1 *Let no two different binders in B bind the same basic name, π_B be a principal ($\mathcal{P}$-)type of $(\!|B|\!)$, and $\mathsf{dom}(\Delta) = \mathsf{fbn}(B)$. Then $\Delta \vdash B$ iff $\Delta \cong \pi_B$.*

The requirement on different binders (which can be achieved by renaming) is not preserved under rewriting because replication can introduce two same-named binders. However, when all binding basic names differ in B_0, then the theorem holds for any successor B_1 of B_0 even when the requirement is not met for B_1. We want to ensure that the derivation of $\Delta \vdash B$ does not assign different types to different bound names. A slightly stronger assumption of Thm. 3.1 simplifies its formulation. The theorem uses principal types and does not necessarily hold for a non-principal $\mathcal{P}$-type π of $(\!|B|\!)$ because π's additional edges not needed to match $(\!|B|\!)$ can preclude $\Delta \cong \pi$.

3.5 Conclusions

We showed a process (Ex. 3.1) that can not be proved communication safe by TPI (Ex. 3.2) but can be proved so by POLY✶ (Ex. 3.3). Thm. 3.1 implies that POLY✶ recognizes safety of all TPI-safe processes. Thus we conclude that POLY✶ is better in recognition of communication safety then TPI. Thm. 3.1 allows to recognize typability in TPI: B is typable in TPI iff $\emptyset \cong \pi_B$. This is computable because a POLY✶ principal type can always be found (for $S_{\mathcal{P}}$ in polynomial time), and checking $\cong$ is easy.

Turner [16, Ch. 5] presents also a polymorphic system for the π-calculus which recognizes B from Ex. 3.1 as safe. However, with respect to our best knowledge, it can not recognize safety of the process "B | s<n, a>.0" which POLY✶ can do. We are not aware of any process that can be recognized safe by Turner's polymorphic system but not by POLY✶. It must be noted, there are still processes which POLY✶ can not prove safe, for example, "a(x).a(y, z).0 | a<o>.a<o, o>.0".

Syntax of Ma *processes:*

$$\begin{array}{rlll}
n \in & \mathsf{AName} & = & \mathsf{Name} \setminus \{\bullet\} \\
N \in & \mathsf{ACapability} & ::= & \varepsilon \mid n \mid \mathsf{in}\ N \mid \mathsf{out}\ N \mid \mathsf{open}\ N \mid N.N' \\
\omega \in & \mathsf{AMessageType} & ::= & \text{definition postponed to Fig. 8} \\
B \in & \mathsf{AProcess} & ::= & \mathsf{0} \mid (B_0 \mathrel{\mathsf{|}} B_1) \mid N\mathtt{[}B\mathtt{]} \mid N.B \mid !B \mid (\nu n{:}\omega)B \mid \\
 & & & \mathtt{<}N_1,\ldots,N_k\mathtt{>} \mid \mathtt{(}n_1{:}\omega_1,\ldots,n_k{:}\omega_k\mathtt{)}.B
\end{array}$$

Rewriting relation of Ma *(≡ is standard defined in TR [9, Fig. 12]):*

$$\begin{array}{rcl}
n\mathtt{[}\mathsf{in}\ m.B_0 \mathrel{\mathsf{|}} B_1\mathtt{]} \mathrel{\mathsf{|}} m\mathtt{[}B_2\mathtt{]} & \to & m\mathtt{[}n\mathtt{[}B_0 \mathrel{\mathsf{|}} B_1\mathtt{]} \mathrel{\mathsf{|}} B_2\mathtt{]} \\
m\mathtt{[}n\mathtt{[}\mathsf{out}\ m.B_0 \mathrel{\mathsf{|}} B_1\mathtt{]} \mathrel{\mathsf{|}} B_2\mathtt{]} & \to & n\mathtt{[}B_0 \mathrel{\mathsf{|}} B_1\mathtt{]} \mathrel{\mathsf{|}} m\mathtt{[}B_2\mathtt{]} \\
\mathsf{open}\ n.B_0 \mathrel{\mathsf{|}} n\mathtt{[}B_1\mathtt{]} & \to & B_0 \mathrel{\mathsf{|}} B_1 \\
\mathtt{(}n_1{:}\omega_1,\ldots,n_k{:}\omega_k\mathtt{)}.B \mathrel{\mathsf{|}} \mathtt{<}N_1,\ldots,N_k\mathtt{>} & \to & B\{n_1 \mapsto N_1,\ldots,n_k \mapsto N_k\}
\end{array}$$

$$\begin{array}{ll}
B_0 \to B_1 \Rightarrow n\mathtt{[}B_0\mathtt{]} \to n\mathtt{[}B_1\mathtt{]} & B_0 \to B_1 \Rightarrow (\nu n{:}\omega)B_0 \to (\nu n{:}\omega)B_1 \\
B_0 \to B_1 \Rightarrow B_0 \mathrel{\mathsf{|}} B_2 \to B_1 \mathrel{\mathsf{|}} B_2 & B_0' \equiv B_0 \ \&\ B_0 \to B_1 \ \&\ B_1 \equiv B_1' \Rightarrow B_0' \to B_1'
\end{array}$$

Figure 7: Syntax and semantics of Tma.

Other π-calculus type systems are found in the literature. Kobayashi and Igarashi [7] present types for the π-calculus looking like simplified processes which can verify properties which are hard to express using shape types (race conditions, deadlock detection) but do not support polymorphism. One can expect applications where Poly✶ is more expressive as well as contrariwise. Shape types, however, work for many process calculi, not just the π-calculus.

4 Shape Types for Mobile Ambients

4.1 Mobile Ambients (Ma)

Mobile Ambients (Ma), introduced by Cardelli and Gordon [3], is a process calculus for representing process mobility. Processes are placed inside named bounded locations called *ambients* which form a tree hierarchy. Processes can change the hierarchy and send messages to nearby processes. Messages contain either ambient names or hierarchy change instructions.

Fig. 7 describes Ma process syntax. Executing a capability consumes it and instructs the surrounding ambient to change the hierarchy. The capability "in n" causes moving into a sibling ambient named n, the capability "out n" causes moving out of the parent ambient n and becoming its sibling, and "open n" causes dissolving the boundary of a child ambient n. In capability sequences, the left-most capability will be executed first.

The constructors "0", "|", ".", "!", and "ν" have standard meanings. Binders contain explicit type annotations (Sec. 4.2 below). The expression $n\mathtt{[}B\mathtt{]}$ describes the process B running inside ambient n. Capabilities can be communicated in messages. $\mathtt{<}N_1,\ldots,N_k\mathtt{>}$ is a process that sends a k-tuple of messages. $\mathtt{(}n_1{:}\omega_1,\ldots,n_k{:}\omega_k\mathtt{)}.B$ is a process that receives a k-tuple of messages, substitutes them for appropriate n_i's in B, and continues as this new process. Free and bound (basic) names are defined like in Meta✶. Processes that are α-

Syntax of TMA *types:*

$$\begin{array}{rlcl} \omega \in & \mathsf{AMessageType} & ::= & \mathsf{Amb}[\kappa] \mid \mathsf{Cap}[\kappa] \\ \kappa \in & \mathsf{AExchangeType} & ::= & \mathsf{Shh} \mid \omega_1 \otimes \cdots \otimes \omega_k \\ \Delta \in & \mathsf{AEnvironment} & = & \mathsf{AName} \to_{\mathrm{fin}} \mathsf{AMessageType} \end{array}$$

Typing rules of TMA*:*

$$\Delta(n) = \omega \Rightarrow \Delta \vdash n : \omega$$
$$\Delta \vdash N : \mathsf{Amb}[\kappa'] \Rightarrow \Delta \vdash \mathsf{in}\ N : \mathsf{Cap}[\kappa]$$
$$\Delta \vdash N : \mathsf{Amb}[\kappa'] \Rightarrow \Delta \vdash \mathsf{out}\ N : \mathsf{Cap}[\kappa]$$
$$\Delta \vdash N : \mathsf{Amb}[\kappa] \Rightarrow \Delta \vdash \mathsf{open}\ N : \mathsf{Cap}[\kappa]$$
$$\Delta \vdash \varepsilon : \mathsf{Cap}[\kappa]$$
$$\Delta \vdash N : \mathsf{Cap}[\kappa]\ \&\ \Delta \vdash N' : \mathsf{Cap}[\kappa] \Rightarrow \Delta \vdash N.N' : \mathsf{Cap}[\kappa]$$

$$\Delta \vdash B : \kappa \Rightarrow \Delta \vdash\ !B : \kappa$$
$$\Delta \vdash \mathsf{0} : \kappa$$
$$\Delta \vdash N : \mathsf{Cap}[\kappa]\ \&\ \Delta \vdash B : \kappa \Rightarrow \Delta \vdash N.B : \kappa$$
$$\Delta \vdash N : \mathsf{Amb}[\kappa]\ \&\ \Delta \vdash B : \kappa \Rightarrow \Delta \vdash N[B] : \kappa'$$
$$\Delta \vdash B_0 : \kappa\ \&\ \Delta \vdash B_1 : \kappa \Rightarrow \Delta \vdash B_0 \mid B_1 : \kappa$$

$$\Delta[n \mapsto \mathsf{Amb}[\kappa']] \vdash B : \kappa \Rightarrow \Delta \vdash (\nu n{:}\mathsf{Amb}[\kappa'])B : \kappa$$
$$\forall i{:}\, 0 < i \leq k\ \&\ \Delta \vdash N_i : \omega_i \Rightarrow \Delta \vdash \langle N_1, \ldots, N_k \rangle : \omega_1 \otimes \cdots \otimes \omega_k$$
$$\Delta[n_1 \mapsto \omega_1, \ldots, n_k \mapsto \omega_k] \vdash B : \omega_1 \otimes \cdots \otimes \omega_k \Rightarrow \Delta \vdash (n_1{:}\omega_1, \ldots, n_k{:}\omega_k).B : \omega_1 \otimes \cdots \otimes \omega_k$$

Figure 8: Syntax of TMA types and typing rules.

convertible are identified. A substitution σ is a finite function from names to messages and its application to B is written $B\sigma$. Fig. 7 also describes structural equivalence and semantics of MA processes. The only thing the semantics does with type annotations is copy them around. We require all processes to be well-scoped w.r.t. conditions S1-3 from Sec. 3.1, and the additional condition (S4) that the same message type is assigned to bound names with the same basic name. Ambients and capabilities where N is not a single name, which the presentation allows for simplicity, are inert and meaningless.

Example 4.1 *In this example, packet ambient* p *delivers a synchronization message to destination ambient* d *by following instructions* x*. As we have not yet properly defined message types, we only suppose* $\omega_{\mathsf{p}} = \mathsf{Amb}[\kappa]$ *for some* κ*.*

$$\begin{array}{l} B = \langle \mathsf{in\ d} \rangle \mid (\nu \mathsf{p}{:}\omega_{\mathsf{p}})(\mathsf{d}[\mathsf{open\ p.0}] \mid (\mathsf{x} : \omega_{\mathsf{x}}).\mathsf{p}[\mathsf{x}.\langle\rangle]) \to \\ \quad (\nu \mathsf{p}{:}\omega_{\mathsf{p}})(\mathsf{d}[\mathsf{open\ p.0}] \mid \mathsf{p}[\mathsf{in\ d}.\langle\rangle]) \to (\nu \mathsf{p}{:}\omega_{\mathsf{p}})(\mathsf{d}[\mathsf{open\ p.0} \mid \mathsf{p}[\langle\rangle]]) \to \mathsf{d}[\langle\rangle] \end{array}$$

4.2 Types for Mobile Ambients (TMA)

An arity mismatch error, like in "<a, b>.0 | (x).in x.0", can occur in polyadic MA. Another communication error can be encountered when a sender sends a capability while a receiver expects a single name. For example "<in a>.0 | (x).out x.0" can rewrite to a meaningless "out (in a).0". Yet another error happens when a process is to execute a single name capability, like in "a.0". Processes which can never evolve to a state with any of the above errors are called *communication safe.* A typed MA introduced by Cardelli and Gordon [4], which we name TMA, verifies communication safety.

TMA assigns an allowed communication topic to each ambient location and ensures that processes respect the topics. Fig. 8 describes TMA type syntax.

Exchange types, which describe communication topics, are assigned to processes and ambient locations. The type Shh indicates silence (no communication). $\omega_1 \otimes \cdots \otimes \omega_k$ indicates communication of k-tuples of messages whose i-th member has the message type ω_i. For $k = 0$ we write $\mathbf{1}$ which allows only synchronization actions <> and (). $\mathsf{Amb}[\kappa]$ is the type of an ambient where communication described by κ is allowed. $\mathsf{Cap}[\kappa]$ describes capabilities whose execution can unleash exchange κ (by opening some ambient). Environments assign message types to free names (via basic names). Fig. 8 also describes the TMA typing rules. Types from conclusions not mentioned in the assumption can be arbitrary. For example, the type of $N[B]$ can be arbitrary provided B is well-typed. It reflects the fact that the communication inside N does not directly interact with N's outside. Existence of some Δ and κ such that Δ does not assign a Cap-type to any free name and $\Delta \vdash B : \kappa$ holds implies that B is communication safe.

Example 4.2 *Take B from Ex. 4.1, $\Delta = \{\mathsf{d} \mapsto \mathsf{Amb}[\mathbf{1}]\}$, and $\omega_{\mathsf{p}} = \mathsf{Amb}[\mathbf{1}]$, and $\omega_{\mathsf{x}} = \mathsf{Cap}[\mathbf{1}]$. We can see that $\Delta \vdash B : \mathsf{Cap}[\mathbf{1}]$ but, for example, $\Delta \not\vdash B : \mathbf{1}$.*

4.3 Instantiation of META✶ to MA

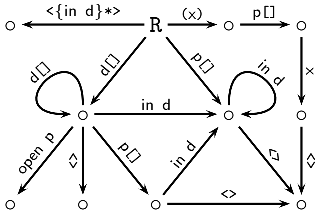
When we omit type annotations, add "0" after output actions, and write capability prefixes always in a right associative manner (like "in a.(out b.(in c.0))"), we see that the MA syntax is included in the META✶ syntax. The following set $\mathcal{A}$ instantiates META✶ to MA.

$$\begin{aligned}
\mathcal{A} = \{\ & \textbf{active}\{\,\mathring{\mathtt{P}}\ \textbf{in}\ \mathring{\mathtt{a}}[\mathring{\mathtt{P}}]\,\},\\
& \textbf{rewrite}\{\,\mathring{\mathtt{a}}[\mathtt{in}\ \mathring{\mathtt{b}}.\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{b}}[\mathring{\mathtt{R}}] \hookrightarrow \mathring{\mathtt{b}}[\mathring{\mathtt{a}}[\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{R}}]\,\},\\
& \textbf{rewrite}\{\,\mathring{\mathtt{a}}[\mathring{\mathtt{b}}[\mathtt{out}\ \mathring{\mathtt{a}}.\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{R}}] \hookrightarrow \mathring{\mathtt{a}}[\mathring{\mathtt{R}}] \mid \mathring{\mathtt{b}}[\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}]\,\},\\
& \textbf{rewrite}\{\,\mathtt{open}\ \mathring{\mathtt{a}}.\mathring{\mathtt{P}} \mid \mathring{\mathtt{a}}[\mathring{\mathtt{R}}] \hookrightarrow \mathring{\mathtt{P}} \mid \mathring{\mathtt{R}}\,\}\ \} \cup\\
\textstyle\bigcup_{k=0}^{\infty} \{\ & \textbf{rewrite}\{\,\mathtt{<}\mathring{\mathtt{M}}_1,\ldots,\mathring{\mathtt{M}}_k\mathtt{>}.\mathring{\mathtt{P}} \mid (\mathring{\mathtt{a}}_1,\ldots,\mathring{\mathtt{a}}_k).\mathring{\mathtt{Q}} \hookrightarrow\ \mathring{\mathtt{P}} \mid \{\mathring{\mathtt{a}}_1 := \mathring{\mathtt{M}}_1,\ldots,\mathring{\mathtt{a}}_k := \mathring{\mathtt{M}}_k\}\,\mathring{\mathtt{Q}}\,\}\ \}
\end{aligned}$$

The **active** rule lets rewriting be done inside ambients. It corresponds to the rule "$B_0 \to B_1 \Rightarrow n[B_0] \to n[B_1]$". Each communication prefix length has its own rule as in the case of the π-calculus. $\mathcal{A}$ defines the calculus $C_{\mathcal{A}}$ and the type system $S_{\mathcal{A}}$.

Communication safety of P can be checked on an $\mathcal{A}$-type as follows. Two edges with the same source labeled by $(a_1, \ldots, a_k)$ and $<b_1, \ldots, b_j>$ with $k \neq j$ indicates an arity mismatch error (but only at active positions). Every label containing $\bullet$ (introduced by a substitution) indicates that a capability was sent instead of a name. Moreover, an edge labeled with a name $a \notin \mathsf{ibn}(P)$ at active position indicates an execution of a single name capability. A type of P not indicating any error proves P's safety. Checking safety this way is easy.

Example 4.3. $C_{\mathcal{A}}$'s equivalent of B from Ex. 4.1 is $P =$ <in d>.0 | νp.(d[open p.0] | (x).p[x.<>.0]). Its principal $\mathcal{A}$-type is displayed on the right. Its root is R and other node names are omitted. Checking the edge labels as described above easily proves P's safety. The edge labeled by x is not a communication error because x is input-bound in P.



4.4 Embedding of TMA in POLY✶

Using the notation from Sec.2.4 we have that C is MA, S_C is TMA, predicates φ are pairs (Δ, κ), and S_C's relation $\rhd B : \varphi$ is $\Delta \vdash B : \kappa$. Moreover $\mathcal{R}$ is $\mathcal{A}$ which was introduced with $C_{\mathcal{A}}$ and $S_{\mathcal{A}}$ in Sec. 4.3. This section provides an embedding which shows how to, for a given B, Δ, and κ, answer the question $\Delta \vdash B : \kappa$ using $S_{\mathcal{A}}$. We stress that it is primarily a theoretical embedding for proving greater expressiveness which is not intended for use in practice.

An encoding $(\!|\cdot|\!)$ of MA processes in META✶, found in TR [9, Fig. 14], is again almost an identity except for the following. (1) Meaningless expressions allowed by MA's syntax are translated using the special name $\bullet$, e.g., "$(\!|\mathsf{in}\ (\mathsf{out}\ \mathsf{a})|\!) = \mathsf{in}\ \bullet$". (2) The encoding erases type annotations which is okay because MA's rewriting rules only copy them around. The type embedding below recovers type information by different means. Prop. 2.1 holds in the given context.

As discussed in Sec. 2.4, we can not translate (Δ, κ) to a shape type with an equivalent meaning because $\vdash$ is preserved under renaming of bound basic names. Nevertheless this becomes possible when we specify the sets of allowed input- and ν-bound basic names and their types. These can be easily extracted from a given process B. An environment Δ^{ν}_{B} (resp. Δ^{in}_{B}) from the top part of Fig. 9 describes ν-bound (resp. input-bound) basic names of B. The definition reflects that ν-bound names in typable processes can only have Amb-types. For a given Δ, B, and κ we construct the shape type $\langle\!|\Delta \cup \Delta^{\nu}_{B}, \Delta^{\mathsf{in}}_{B}, \kappa|\!\rangle$ such that $\Delta \vdash B : \kappa$ iff $\vdash (\!|B|\!) : \langle\!|\Delta \cup \Delta^{\nu}_{B}, \Delta^{\mathsf{in}}_{B}, \kappa|\!\rangle$. The construction needs to know which names are input-bound and thus they are separated from the other names. The well-scopedness rules S1-4 ensure that there is no ambiguity in using only basic names to refer to typed names in a process. The type information I (Fig. 9, 2nd part) collects what is needed to construct a shape type. For $I = (\Delta \cup \Delta^{\nu}_{B}, \Delta^{\mathsf{in}}_{B}, \kappa)$ we define Δ_I, Δ^{in}_{I}, and κ_I such that Δ_I describes types of all names in Δ and B, and Δ^{in}_{I} describes types of B's input-bound names, and κ_I is simply κ.

Example 4.4 *Δ, B, and κ from the previous examples (Ex. 4.1 and Ex. 4.2) give us $I = (\Delta \cup \Delta^{\nu}_{B}, \Delta^{\mathsf{in}}_{B}, \mathsf{Cap}[\mathbf{1}])$ and we have:*

$$\Delta \cup \Delta^{\nu}_{B} = \{\mathsf{d} \mapsto \mathsf{Amb}[\mathbf{1}], \mathsf{p} \mapsto \mathsf{Amb}[\mathbf{1}]\} \quad \Delta^{\mathsf{in}}_{I} = \{\mathsf{x} \mapsto \mathsf{Cap}[\mathbf{1}]\} \quad \Delta_I = \Delta \cup \Delta^{\nu}_{B} \cup \Delta^{\mathsf{in}}_{I}$$

The main idea of the construction of the shape type $\langle\!|I|\!\rangle$ from I is that $\langle\!|I|\!\rangle$ contains exactly one node for every exchange type of some ambient location, that is, one node for the top-level type κ_I, and one node for κ' whenever $\mathsf{Amb}[\kappa']$ is in I. The top-level type corresponds to the shape type root. Each node corresponding to some κ has self-loops which describe all capabilities and communication actions which a process of the type κ can execute. When $\Delta_I(\mathsf{d}) = \mathsf{Amb}[\mathbf{1}]$ then every node would have a self-loop labeled by "$\mathsf{in}\ \mathsf{d}$" because in-capabilities can be executed by any process. On the other hand only the node corresponding to $\mathbf{1}$ would allow "$\mathsf{open}\ \mathsf{d}$" because only processes of type $\mathbf{1}$ can legally execute it. Finally, following an edge labeled with "$\mathsf{d}[]$" means entering d. Thus the edge has led to the node χ_{d} that corresponds to $\mathbf{1}$. In the above example, the shape graph would contain edges labeled with "$\mathsf{d}[]$" from any node to χ_{d}.

Extraction of types of bound names:

$$\Delta^{\mathsf{in}}_B(a) = \omega \text{ iff } B \text{ has a subprocess } \texttt{(}\ldots, a^i : \omega, \ldots\texttt{)}.B_0$$

$$\Delta^{\nu}_B(a) = \omega \text{ iff } \omega = \mathsf{Amb}\,\texttt{[}\kappa\texttt{]} \;\&\, B \text{ has a subprocess } (\nu a^i : \omega)B_0$$

Type information:

$$I \in \mathsf{TypeInfo} = \mathsf{AEnvironment} \times \mathsf{AEnvironment} \times \mathsf{AExchangeType}$$

For a given $I = (\Delta_0, \Delta_1, \kappa)$ we write Δ_I for $\Delta_0 \cup \Delta_1$, and Δ^{in}_I for Δ_1, and κ_I for κ.

Set of nodes of a shape graph (and correspondence functions):

$$\mathsf{types}_I = \{\kappa_I\} \cup \{\kappa \colon \mathsf{Amb}\,\texttt{[}\kappa\texttt{]} \in \mathsf{rng}(\Delta_I)\} \qquad \mathsf{nodeof}_I = \mathsf{typeof}_I^{-1}$$

Let nodes_I be an arbitrary but fixed set of nodes such that there exist the bijection typeof_I from nodes_I into types_I.

Action types describing legal capabilities:

$\mathsf{namesof}_I(\omega) = \{a \colon \Delta_I(a) = \omega\} \qquad \mathsf{allowedin}_I(\kappa) = \mathsf{moves}_I \cup \mathsf{opens}_I(\kappa) \cup \mathsf{comms}_I(\kappa)$

$\mathsf{moves}_I = \{\mathsf{in}\ a, \mathsf{out}\ a \colon \exists\kappa.\, a \in \mathsf{namesof}_I(\mathsf{Amb}\,\texttt{[}\kappa\texttt{]}\,)\}$
$\mathsf{opens}_I(\kappa) = \{\mathsf{open}\ a \colon a \in \mathsf{namesof}_I(\mathsf{Amb}\,\texttt{[}\kappa\texttt{]}\,)\} \cup \mathsf{namesof}_I(\mathsf{Cap}\,\texttt{[}\kappa\texttt{]}\,)$
$\mathsf{msgs}_I(\mathsf{Amb}\,\texttt{[}\kappa\texttt{]}\,) = \mathsf{namesof}_I(\mathsf{Amb}\,\texttt{[}\kappa\texttt{]}\,)$
$\mathsf{msgs}_I(\mathsf{Cap}\,\texttt{[}\kappa\texttt{]}\,) = \mathsf{namesof}_I(\mathsf{Cap}\,\texttt{[}\kappa\texttt{]}\,) \cup \{(\mathsf{moves}_I \cup \mathsf{opens}_I(\kappa))\texttt{*}\}$
$\mathsf{comms}_I(\mathsf{Shh}) = \emptyset \qquad \mathsf{comms}_I(\omega_1 \otimes \cdots \otimes \omega_k) = \{\texttt{<}\mu_1, \ldots, \mu_k\texttt{>} \colon \mu_i \in \mathsf{msgs}_I(\omega_i)\} \cup$
$\qquad\qquad \{\texttt{(}a_1, \ldots, a_k\texttt{)} \colon \Delta^{\mathsf{in}}_I(a_i) = \omega_i \;\&\; (i \neq j \Rightarrow a_i \neq a_j)\}$

Construction of shape predicates:

$\langle\!| I |\!\rangle = \langle \langle\!| I |\!\rangle, \mathsf{nodeof}_I(\kappa_I)\rangle \quad \langle\!| I |\!\rangle = \{\chi \xrightarrow{\alpha} \chi \colon \alpha \in \mathsf{allowedin}_I(\mathsf{typeof}_I(\chi)) \,\&\, \chi \in \mathsf{nodes}_I\} \cup$
$\qquad \{\chi \xrightarrow{a\,\texttt{[]}} \chi' \colon a \in \mathsf{namesof}_I(\mathsf{Amb}\,\texttt{[}\mathsf{typeof}_I(\chi')\texttt{]}\,) \,\&\, \chi, \chi' \in \mathsf{nodes}_I\}$

Figure 9: Construction of Poly✶ type embedding.

The construction starts by building the node set of a shape predicate (Fig. 9, 3rd part). All the exchange types of ambient locations are gathered in the set types_I. These types are put in bijective correspondence with the set nodes_I.

Example 4.5 *Our example gives* $\mathsf{types}_I = \{\mathsf{Cap}\,\texttt{[}\mathbf{1}\texttt{]}\,, \mathbf{1}\}$. *Let us take* $\mathsf{nodes}_I = \{\texttt{R}, \texttt{1}\}$ *and define the bijections such that* $\mathsf{nodeof}_I(\mathsf{Cap}\,\texttt{[}\mathbf{1}\texttt{]}\,) = \texttt{R}$ *and* $\mathsf{nodeof}_I(\mathbf{1}) = \texttt{1}$.

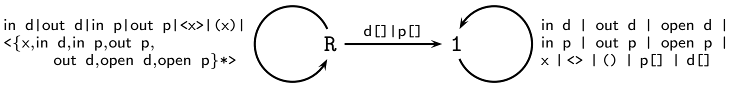
The 4th part of Fig. 9 defines some auxiliary functions. The set $\mathsf{namesof}_I(\omega)$ contains all basic names declared with the type ω by I. The set $\mathsf{allowedin}_I(\kappa)$ contains all Poly✶ action types which describe (translations of) all capabilities and action prefixes which are allowed to be legally executed by a process of the type κ. The set $\mathsf{allowedin}_I(\kappa)$ consists of three parts: moves_I, $\mathsf{opens}_I(\kappa)$, and $\mathsf{comms}_I(\kappa)$. The action types in moves_I describe all in/out capabilities constructible from ambient basic names in I. The set does not depend on κ because in/out capabilities can be executed by any process. The set $\mathsf{opens}_I(\kappa)$ describe open-capabilities which can be executed by a process of the type κ. The second part of $\mathsf{opens}_I(\kappa)$ describes names of the type $\mathsf{Cap}\,\texttt{[}\kappa\texttt{]}$ which might be instantiated to some executable capabilities. The set $\mathsf{comms}_I(\kappa)$ describes communication actions which can be executed by a process of the type κ. Its first part describes output- and the second input-actions. The auxiliary set $\mathsf{msgs}_I(\omega)$ describes all messages of the type ω constructible from names in I.

Example 4.6 *Relevant sets for our example are:*

$$\begin{array}{ll}
\mathsf{namesof}_I(\mathsf{Amb}\texttt{[}\mathbf{1}\texttt{]}) = \{\mathsf{d},\mathsf{p}\} & \mathsf{opens}_I(\mathbf{1}) = \{\mathsf{open\ d},\mathsf{open\ p},\mathsf{x}\} \\
\mathsf{namesof}_I(\mathsf{Cap}\texttt{[}\mathbf{1}\texttt{]}) = \{\mathsf{x}\} & \mathsf{opens}_I(\mathsf{Cap}\texttt{[}\mathbf{1}\texttt{]}) = \emptyset \\
\mathsf{comms}_I(\mathbf{1}) = \{\texttt{<>},\texttt{()}\} & \mathsf{moves}_I = \{\mathsf{in\ d},\mathsf{in\ p},\mathsf{out\ d},\mathsf{out\ p}\} \\
\multicolumn{2}{l}{\mathsf{comms}_I(\mathsf{Cap}\texttt{[}\mathbf{1}\texttt{]}) = \{\texttt{<x>},\texttt{<\{}\mathsf{in\ d},\mathsf{in\ p},\mathsf{out\ d},\mathsf{out\ p},\mathsf{open\ d},\mathsf{open\ p},\mathsf{x}\texttt{\}*>},\texttt{(x)}\}}
\end{array}$$

The bottom part of Fig. 9 constructs the shape graph $\langle\!| I |\!\rangle$ and the shape predicate $\langle\!| I |\!\rangle$ from I. The first part of $\langle\!| I |\!\rangle$ describes self-loops of χ which describe actions allowed to be executed by a process of $\mathsf{typeof}_I(\chi)$. The second part of $\langle\!| I |\!\rangle$ describe transitions among nodes. Any edge labeled by "$a\,\texttt{[]}$" always leads to the node which corresponds to the exchange type allowed inside a.

Example 4.7 *The resulting shape predicate $\langle\!| I |\!\rangle = \langle G, \mathtt{R}\rangle$ in our example is as follows. We merge edges with the same source and destination using "*|*".*



Correctness of the translation is expressed by Thm. 4.1. The assumptions ensure that no ν-bound name is mentioned by Δ or has a Cap-type assigned by an annotation. Here we just claim that $\langle\!| I |\!\rangle$ is always an $\mathcal{A}$-type.

Theorem 4.1 *Let* $\mathsf{dom}(\Delta) \cap \mathsf{nbn}(B) = \emptyset$ *and* $\mathsf{dom}(\Delta^{\nu}_{B}) = \mathsf{nbn}(B)$. *Then it holds that* $\Delta \vdash B : \kappa$ *if and only if* $\vdash (\!| B |\!) : \langle\!| (\Delta \cup \Delta^{\nu}_{B}, \Delta^{\mathsf{in}}_{B}, \kappa) |\!\rangle$.

4.5 Conclusions

We embedded Tma's typing relation in $S_{\mathcal{A}}$ (Sec. 4.4) and showed how to recognize communication safety in $S_{\mathcal{A}}$ directly (Sec. 4.3). The type $\langle\!| I |\!\rangle$ constructed in Sec. 4.4 can also be used to prove the safety of B. But then, it follows from the properties of principal types, that the safety of B can be recognized directly from its principal $\mathcal{A}$-type. Thus any process proved safe by Tma can be proved safe by $S_{\mathcal{A}}$ on its own.

Some processes are recognized safe by $S_{\mathcal{A}}$ but not by Tma. For example, "(x : ω).x.0 | <in a>" is not typable in Tma but it is trivially safe. Another examples show polymorphic abilities of shape types, for example, the $C_{\mathcal{A}}$ process

!(x, y, m).x[in y.<m>.0] | <p, a, c>.0 | a[open p.0] | <q, b, in a>.0 | b[open q.0]

can be proved safe by Poly✶ but it constitutes a challenge for Tma-like non-polymorphic type systems. We are not aware of other type systems for Ma and its successors that can handle this kind of polymorphism.

The expressiveness of shape types $\langle\!| I |\!\rangle$ from Sec. 4.4 can be improved. In subsequent work [1], Cardelli, Ghelli, and Gordon define a type system which can ensure that some ambients stay immobile or that their boundaries are never dissolved. This can be achieved easily by removing appropriate self loops of nodes. We can also assign nodes to (groups of) ambients instead of exchange types. This gives us similar possibilities as another TMA successor [2]. Moreover, we can use shape type polymorphism to express location-dependent properties of ambients, like that ambient a can be opened only inside ambient b.

5 Conclusions and Future Work

We discussed already the contributions (Sec. 1.1, 2.5). Conclusions for the embeddings were given separately (Sec. 3.5, 4.5). Future work is as follows. For extensions, priorities are better handling of choice (e.g., because of its use in biological system modeling), and handling of **rec** which is in many calculi more expressive than replication and better describes recursive behavior. Moreover we would like to generalize actions so that calculi with structured messages, like the Spi calculus [5], can be handled. For applications, we would like to (1) relate shape types with other systems which also use graphs to represent types [18, 10], and (2) to study the relationship between shape types and session types [6].

References

[1] L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility types for mobile ambients. In *ICALP*, volume 1644 of *LNCS*, pages 230–239, July 1999.

[2] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *IFIP TCS*, volume 1872 of *LNCS*, pages 333–347, Aug. 2000.

[3] L. Cardelli and A. D. Gordon. Mobile ambients. In *FoSSaCS*, volume 1378 of *LNCS*, pages 140–155, 1998.

[4] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *POPL*, pages 79–92, 1999.

[5] M. A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. & Comp.*, 148(1):1–70, Jan. 1999.

[6] K. Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523, 1993.

[7] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *POPL*, pages 128–141, 2001.

[8] J. Jakubův. *A Second Year Report.* Heriot-Watt Univ., MACS., 2009. Available at http://www.macs.hw.ac.uk/˜jj36.

[9] J. Jakubův and J. B. Wells. The expressiveness of generic process shape types. Technical Report HW-MACS-TR-0069, Heriot-Watt Univ., July 2009.

[10] B. König. Generating type systems for process graphs. In *CONCUR*, volume 1664 of *LNCS*, pages 352–367, 1999.

[11] H. Makholm and J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. Technical Report HW-MACS-TR-0022, Heriot-Watt Univ., Nov. 2004.

[12] H. Makholm and J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. In *ESOP*, volume 3444 of *LNCS*, pages 389–407, 2005.

[13] R. Milner. *Communicating and Mobile Systems: The π-Calculus.* Cambridge Press, 1999.

[14] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Inf. & Comp.*, 100(1):1–77, Sept. 1992.

[15] F. Nielson, H. R. Nielson, C. Priami, and D. Rosa. Control flow analysis for bioambients. *ENTCS*, 180(3):65–79, 2007.

[16] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation.* PhD thesis, Uni. of Edinburgh, 1995. Rep. ECS-LFCS-96-345.

[17] J. B. Wells. The essence of principal typings. In *ICALP*, volume 2380 of *LNCS*, pages 913–925, 2002.

[18] N. Yoshida. Graph types for monadic mobile processes. In *FSTTCS*, volume 1180 of *LNCS*, pages 371–386, 1996.